Vector Reduction/Transformation Operators

ROSCOE A. BARTLETT
BART G. VAN BLOEMEN WAANDERS
MICHAEL A. HEROUX
Sandia National Laboratories, Albuquerque NM 87185 USA

Development of flexible linear algebra interfaces is an increasingly critical issue. Efficient and expressive interfaces are well established for some linear algebra abstractions, but not for vectors. Vectors differ from other abstractions in the diversity of necessary operations, sometimes requiring dozens for a given algorithm (e.g. interior-point methods for optimization). We discuss a new approach based on operator objects that are transported to the underlying data by the linear algebra library implementation, allowing developers of abstract numerical algorithms to easily extend the functionality regardless of computer architecture, application or data locality/organization. Numerical experiments demonstrate efficient implementation.

Categories and Subject Descriptors: ... [...]: ...

General Terms: Algorithms, Design, Performance, Standardization

Additional Key Words and Phrases: Optimization, Object-Orientation, Vectors, Interfaces, ...

1. INTRODUCTION

Many mathematical algorithms are concerned with the construction and manipulation of vectors and vector spaces. Typical situations include the construction of an orthogonal basis of vectors, or computing search directions in a multi-dimensional space. A common characteristic of these types of algorithms is that remarkably little detailed information about the vectors is required in order to implement the abstract numerical algorithm (ANA). We typically do not need to know if the vector is stored on a serial computer or partitioned across a distributed memory computer. In fact, the storage of the vector data, and even the actual mathematical computation involving the vectors can be done remotely from the computer that is executing the ANA.

What an ANA does require is that vectors be compatible with each other, and that certain operations such as vector norms, scalings and transformations can be applied to the vectors. Furthermore, some classes of ANAs require a mechanism for extending functionality, since the ANA has too many specialized vector operations for a general-purpose vector library to implement *a priori*.

This ability to separate the vector functionality needed by an ANA from the

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

details of the vector implementation is widely known, and its importance cannot be overstated. Sophisticated ANA's are a challenge to implement, independent of implementation details of how vectors are stored or how vector operations are performed. The real value of a good ANA is the careful attention to details such as how parameters are selected to maintain orthogonality of a subspace or to avoid stagnation or divergence. This observation has two major implications:

(1) The robustness of an ANA is essentially independent of the details of how vectors are stored and computed.
(2) Sophisticated ANA's need a vector interface that is abstract and easily extended.

Numerous abstract vector interfaces and concrete vector implementations have been developed [Gockenbach and Symes ; Heroux et al. ; Balay et al. ; Pozo, R. ; 1996; Lumsdanie and Siek 1998b]. At the same time, none of the existing approaches have succeeded in maximizing the potential of the separation of interface and implementation. In all existing approaches there are restrictions on the location of data or the efficient extension of functionality, or both.

In this paper we present a simple and elegant mechanism that allows a maximum separation of vector functionality from the details of implementation for a broad class of vector reduction and transformation operations. This mechanism allows data storage and computation to be completely separated from the ANA. It also allows straightforward extension of vector functionality and can easily be incorporated into existing vector libraries. Complete source code along with Doxygen[1] generated documentation for all of the examples described in this paper can be found at `WEBSITE`[2]

2. BACKGROUND

We subdivide a typical numerical simulation code into three major components to differentiate the ANA from other components that also require interfaces for linear algebra operations (Figure 1). The first category is application (APP) software in which the underlying data is defined for the problem. This could be something as simple as the right hand side and matrix coefficients of a single linear system or as complex as a finite element method for a 3-D nonlinear PDE-constrained optimization problem. The second category is linear algebra library (LAL) software that implements basic linear algebra operations [Demmel 1997; Anderson et al. 1995; Blackford et al. 1997; Tuminaro et al. 1999; Balay et al. ; Heroux et al.]. These types of software include primarily matrix-vector multiplication, the creation of a preconditioner (e.g. ILU), and may even include several different types of linear solvers. The third category is ANA software that drives the main solution process and includes such algorithms as iterative methods for linear and nonlinear systems; explicit and implicit methods for ODEs and DAEs; and nonlinear programming (NLP) solvers [Nocedal and Wright 1999]. There are many example sofware packages [Balay et al. ; Tuminaro et al. 1999; Heroux et al. ; Byrne and Hindmarsh 1999; Benson et al.] that contain ANA software.

[1] `www.doxygen.org`
[2] `WEBSITE = http://software.sandia.gov/RTOp`

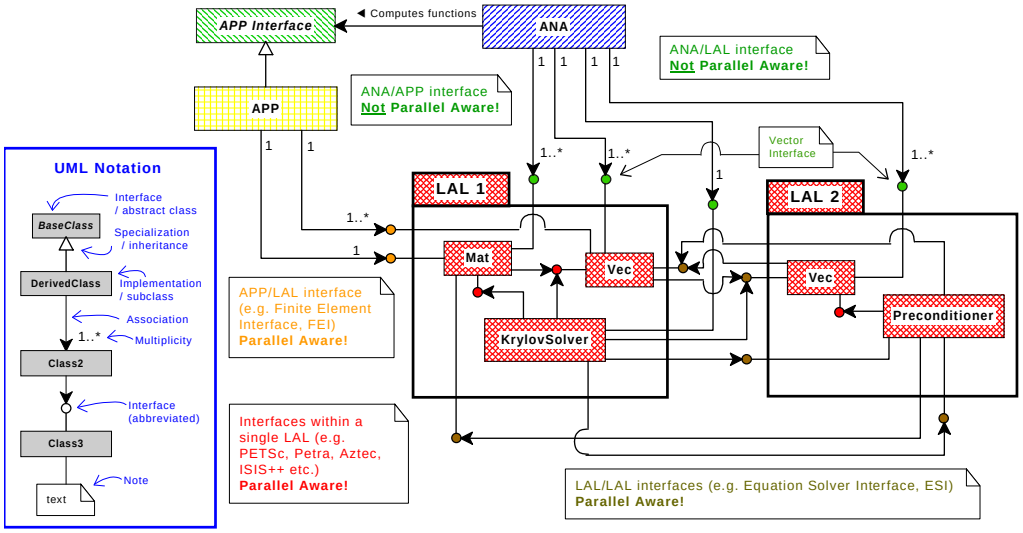

Fig. 1. UML class diagram : Interfaces between abstract numerical algorithm (ANA), linear algebra library (LAL), and application (APP) software.

Multiple interfaces can be identified between ANA, LAL, and APP software components. Although we are interested in the ANA-LAL connection, other interfaces (e.g. APP-LAL [Clay et al. 1999], LAL-LAL [Sandia National Labs 2001]) are required for the makeup of numerical codes, and have different functional requirements. The purpose of the APP-LAL interface is to allow APP software to fill vector and matrix objects with residuals and gradients that define the underlying mathematical problem. This interface needs to be fairly intimate and details such as global data distribution maps (in a parallel application for example) must be exposed by the interface. In some cases, more than one LAL may be used together which requires a LAL-LAL interface. For example, the preconditioner from one LAL may be used with the matrix and vector objects from another LAL in the solution of a linear system using an iterative linear solver. The LAL-LAL interfaces are similar to the APP-LAL interfaces in that they must also be fairly detailed and expose information such as data distribution on a parallel computer. For instance, for a parallel matrix to apply itself to a vector from another LAL, the vector must expose its map of local to global elements and give explicit access to the local vector data. An even more intimate interface must be exposed by a parallel matrix object in order to construct a preconditioner from another LAL.

The focus of our work is the ANA-LAL interface because the realization of this approach can potentially have considerable software development and code efficiency impact. The purpose of the ANA-LAL interface is to provide numerical algorithms with appropriate linear algebra functionality, preferably independent of data mapping and without the involvement of the LAL developer. The central idea is based on operator objects that are transported to the underlying data by the linear algebra library implementation where they are applied on an element-wise

basis. This approach provides developers of abstract numerical algorithms with the ability to easily extend the functionality of a vector interface regardless of computer architecture, application, data locality or organization of the underlying data. The design of the ANA-LAL interface must consider the diversity of application areas, computing environments, and configurations. For example, in a large-scale seismic inversion problem, gigabytes of data are stored "out-of-core" and are repetitively read from disk, operated on, and then written back to disk. Alternatively, the main focus for large-scale scientific computing has been to write SPMD[3] software that runs on large parallel computers, and more recently, client-server and grid computing are being considered where multiple resources may be used to solve a single coupled problem. Issues associated with these different data mappings and computing environments are addressed through our new approach.

Trying to abstract the details of linear algebra data structures and computations away from ANAs is not a new or recent idea. The general specification of the ANA-LAL and ANA-APP interfaces described in [Heinkenschloss and Vicente 1999] allows the development of flexible ANA software. However, this specification does not deal with the issues associated with the locality and mapping of vector data. In addition, none of the current vector-interface approaches (see Section 3.2) provide a sufficient means by which ANA and LAL software can be adequately decoupled so that the work required to glue a particular ANA to a LAL is sufficiently low. Here we describe a new approach by which a developer of an ANA can in fact unilaterally add the implementation for a new vector operation and then have it automatically supported by any LAL implementation for any runtime configuration. This is possible through a specification for user-defined vector operators and the addition of a single method to a vector interface that every LAL implementation can easily support.

3. VECTORS IN NUMERICAL SOFTWARE AND CHALLENGES IN DEVELOPING ABSTRACT INTERFACES

Vectors provide the primary foundation for the ANA-LAL and ANA-APP interfaces. Beyond transporting vector objects back and forth to the APP and LAL interfaces, ANA also need to perform various vector reduction (e.g. norms, dot products) and transformation (e.g. vector addition, scaling) operations. In addition, many specialized "nonstandard" vector operations must be performed. Examples of non-standard operations are presented below to help motivate our design, followed by a discussion of how current and established approaches would attempt to handle these non-standard operations.

3.1 Variety of vector operations needed

Perhaps the primary distinction between vectors and other linear algebra objects is the large number of non-standard operations that a complex ANA requires. In addition to the 15 BLAS [J. J. Dongarra and J. Du Croz and S. Hammarling and R. J. Hanson 1988] operations, many other types of operations need to be performed. For example, some of the nonstandard operations an optimization algorithm (e.g. OOQP [Gertz and Wright 2001]) may perform are:

[3]Single Program, Multiple Data

$$y_i = \begin{cases} y^{\min} - y_i & \text{if } y_i < y^{\min} \\ y^{\max} - y_i & \text{if } y_i > y^{\max} \\ 0 & \text{if } y^{\min} \le y_i \le y^{\max} \end{cases} \quad \text{for } i = 1...n, \tag{1}$$

$$\alpha \leftarrow \{\max \alpha \ : \ x + \alpha d \ge \beta\}, \tag{2}$$

$$\gamma \leftarrow (x + \alpha p)^T (y + \alpha q). \tag{3}$$

The interior-point NLP algorithm described in [Dennis et al. 1998] performs several more unusual vector operations, such as

$$d_i \leftarrow \begin{cases} (b-u)_i^{1/2} & \text{if } w_i < 0 \text{ and } b_i < +\infty \\ 1 & \text{if } w_i < 0 \text{ and } b_i = +\infty \\ (u-a)_i^{1/2} & \text{if } w_i \ge 0 \text{ and } a_i > -\infty \\ 1 & \text{if } w_i \ge 0 \text{ and } a_i = -\infty \end{cases} \quad \text{for } i = 1...n. \tag{4}$$

3.2 Current approaches to developing interfaces for vectors and vector operations

Currently there are three established approaches to abstracting vectors from ANA software that may be used to address special and unusual vector operations. Below we describe each of these approaches and discuss their limitations.

The first approach (I) is to allow an ANA to access the vector data in some controlled way which enables the ANA to perform required operations. This is by far the most common approach and in the case of parallel numerical codes using SPMD this is currently the preferred method [Heroux et al. ; Balay et al. ; Clay et al. 1999]. This approach however assumes that vector data is readily available in every process where the ANA runs. Otherwise, moving large segments of vector data to processes where the ANA is running can cause considerable inefficiencies. For instance, in the case of a client-server architecture, copies of vector data would have to be communicated from the client to the server causing considerable inefficiencies. Approach I potentially provides for an efficient development environment, provided data movement is not an issue and assuming the ANA algorithms do not need to be re-used in other computing environments. Even in an SPMD environment, this approach is not without difficulties (e.g. ghost elements and reduction operations).

More recently, a second approach (II) has been used where each specific ANA defines its own customized abstract LAL interface and then leaves it up to the end user to provide the implementations [Lumsdanie and Siek 1998a; Gertz and Wright 2001; Cai 1999]. ITL [Lumsdanie and Siek 1998a] uses the C++ template mechanism and requires compile-time polymorphism while OOQP [Gertz and Wright 2001] and DiffPack [Cai 1999] use C++ classes and virtual functions and allow for runtime polymorphism. While this approach is more flexible than approach I and abstracts away the data mapping issues, it simply passes the interfacing problem on to the end user, who is forced to implement the required operations given an existing LAL. For example, OOQP includes more than 30 vector operations,

many of which must be implemented from scratch for a new LAL or computing configuration.

Finally, a third approach (III) constructs a general linear algebra interface that tries to anticipate what fundamental or "primitive" operations will be needed. An ANA is expected to implement more specialized operations by stringing together a set of primitives. In theory, this approach allows ANA and LAL software to be developed and maintained independently and be used together with very little extra work. Such an approach was taken by the designers of the Hilbert Class Library (HCL) version 1.0 [Gockenbach and Symes] and was originally motivated by out-of-core data sets even though is applicable to any computing environment. Even though approach III is most promising, it still has three primary shortcomings. First, stringing together a set of primitive operations requires temporary copies and can create an inefficient implementation. Second, this approach requires the standardization of the primitive operations that are part of the interface. Consequently, the developer of an ANA can not add a new method to an existing LAL interface and expect it to be automatically supported by LAL implementations. Third, non-standard operations are often difficult to develop through the use of a finite number of primitives.

To demonstrate how a series of primitive vector operations can be used to implement a more specialized operation, consider the vector reduction operation (2). This operation could be performed with six temporary vectors $u, v, w, y, z \in \mathbb{R}^n$ and the following six primitive vector operations:

$$\begin{array}{l} -x_i \rightarrow u_i, \quad u_i + \beta \rightarrow v_i, \quad v_i/d_i \rightarrow w_i, \quad 0 \rightarrow y_i, \quad max\left\{w_i, y_i\right\} \rightarrow z_i, \\ min\left\{z_i, i = 1...n\right\} \rightarrow \alpha. \end{array} \tag{5}$$

Many other vector operations can be performed using primitives. However, it is difficult to see how operations like (1) and (4) could be implemented with general purpose primitive vector operations. A large number of primitive operations need to be included in a generic vector interface in order to implement most of the required vector operations. For example, the vector interface in HCL 1.0 contains more than 50 operations and still can not accommodate some of the above example vector/array operations.

Another problem is that, in a parallel program, stringing primitives together can result in a serial bottleneck. For instance, in ISIS++ [Clay et al. 1999], the combined reduction operation (6) was implemented for the quasi-minimum-residual (QMR) iterative solver.

$$\left\{\alpha, \gamma, \xi, \rho, \varepsilon\right\} \leftarrow \left\{(x^T x)^{1/2}, (v^T v)^{1/2}, (w^T w)^{1/2}, w^T v, v^T t\right\} \tag{6}$$

All five of the reduction operations in (6) can be performed in one pass through the vector data (four vector reads) and one global reduction. In contrast, if we must rely on primitive LAL methods for computing each reduction separately, we would need at least seven vector reads and five global reductions. It is certainly possible to add an operation like (6) to a LAL, but such an ANA-specialized operation can not

be added to a generic LAL interface without the direct support of the developers of all the LALs used with the ANA.

4. VECTOR REDUCTION/TRANSFORMATION OPERATORS

4.1 Introduction to vector reduction/transformation operations

Our design addresses all the above described limitations associated with approaches I, II, and III. The key design strategy consists of passing user-defined operations to vector objects and having the vector implementations apply the operations to the vector data. ANA developers are therefore not limited to the use of primitives, can freely develop their vector operator implementations, and do not have to depend on temporary copies. In addition to an efficient implementation, this approach is also independent of the underlying data mapping of the vectors. The design allows ANA developers to create any vector reduction/transformation operator (RTOp) that is equivalent to the following element-wise operators:

$$\text{element-wise transformation}: \quad op_T(i, v_i^0 \dots v_i^{p-1}, z_i^0 \dots z_i^{q-1}) \rightarrow z_i^0 \dots z_i^{q-1}, \tag{7}$$

$$\text{element-wise reduction}: \quad op_R(i, v_i^0 \dots v_i^{p-1}, z_i^0 \dots z_i^{q-1}) \rightarrow \beta, \tag{8}$$

$$\text{reduction of intermediates}: \quad op_{RR}(\hat{\beta}, \tilde{\beta}) \rightarrow \tilde{\beta}, \tag{9}$$

where $v^0 \dots v^{p-1} \in \mathbb{R}^n$ are p non-mutable input vectors; $z^0 \dots z^{q-1} \in \mathbb{R}^n$ are q mutable input/output vectors; and β is a reduction target object which may be a simple scalar, a more complex non-scalar (e.g. $\{\alpha, \gamma, \xi, \rho, \epsilon\}$) or NULL. In the most general case, the ANA can define an operator that will simultaneously perform multiple reduction and transformation operations involving a set of vectors. Simpler operations can be formed by setting $p = 0$, $q = 0$ or $\beta =$ NULL. For example, reduction operations over one vector argument, such as vector norms ($||v||$), are defined with $p = 1$, $q = 0$ and $\beta = \{\text{scalar}\}$. With this design, all of the standard BLAS operations, the example vector operations in (1)–(6) and many more vector operators can be expressed. The key to optimal performance is that the vector implementation applies (7) and (8) together on an entire set of sub-vectors (for elements $i = a \dots b$) at once

$$op(a, b, v_{a:b}^0 \dots v_{a:b}^{p-1}, z_{a:b}^0 \dots z_{a:b}^{q-1}, \beta) \rightarrow z_{a:b}^0 \dots z_{a:b}^{q-1}, \beta. \tag{10}$$

In this way, as long as the size of the sub-vectors is sufficiently large, the cost of performing a function call to invoke the operator will be insignificant compared to the cost of performing the computations within the operator. In a parallel distributed vector, $op(\dots)$ is applied to the local sub-vectors on each processor. The only communication between processors is to reduce the intermediate reduction objects $op(\hat{\beta}, \tilde{\beta}) \rightarrow \tilde{\beta}$ (unless $\beta =$ NULL, then no communication is required). It is important to understand that it is the vector implementation that decides how to best segment the vector data into chunks that are passed to the user-defined operator which results in the most efficient implementation possible.

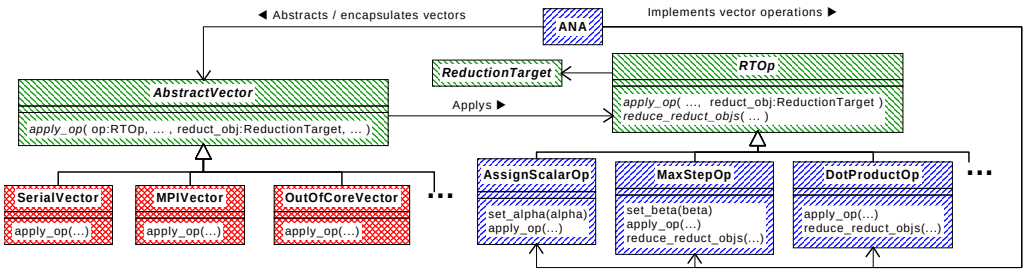

Fig. 2. UML class diagram : vector reduction/transformation operators

On most machines, the dominant cost of performing a vector operation is the movement of data to and from main memory [Demmel 1997]. This is especially true for out-of-core vectors. Therefore, performing multiple operations on a vector at the same time such as in (6) will be faster than performing them separately in most computing environments.

It is important to note that the type of element-wise operators described in (7)–(9) can not be used to implement general linear and nonlinear vector operators. For example, a general linear operator A

$$z = op(v) = Av$$

computes the ith element of the output vector z as a linear combination of perhaps all of the right-hand-side vector elements in v. In a distributed-memory environment, this requires careful handling of vector data which results in the use of ghost elements and the targeted communication of potentially large amounts of vector data. For this reason, the element-wise operators defined in (7) explicitly state this element-wise requirement. As is clearly seen in (7), the ith element in the input/output vectors z can only be computed using information from the ith elements in the vectors v and z and from no other vector elements. It is impossible to design an efficient operator interface that allows non-element-wise transformations that does not also require a detailed knowledge of the computing environment, the layout of vector data and functionality for communicating vector data.

4.2 An object-oriented design for reduction and transformation operators

Here an object-oriented [Booch et al. 1999; Gamma et al. 1995] design for vector reduction/transformation operators is presented which is based on the "Visitor" pattern [Gamma et al. 1995]. Figure 2 shows the general structure of the design. At the core of the design is an interface for vector operators called `RTOp` for which different ANA-specific operator types can be implemented. This operator interface includes methods for the operations (9) and (10). A vector interface `AbstractVector` includes a method that accepts user-defined `RTOp` operator objects (see `WEBSITE` for the special behavior of this method). A vector implementation applies operators in an appropriate manner and returns reduction objects (if non-NULL). Examples of a few different concrete vector subclasses are shown in Figure 2. The reason that this design pattern is called "Visitor" is that an "ObjectStructure" takes a client's user-defined visitor object and then decides how this visitor object will visit

all of the "Elements" containing the data. The key is that the client's user-defined operations are taken to the data in a transparent way, the data is not presented to the client (as in the "Iterator" design pattern).

The mechanism by which a vector implementation applies a user-defined operator depends on the computing environment. The following three scenarios show how an ANA code can be used with the same operator implementations in three different computing configurations: (i) out-of-core, (ii) SPMD, and (iii) client-server. The ANA code does not need to be recompiled for any scenario and the LAL implementations can be changed at run time.

(i) For out-of-core data sets (Figure 3), the vector implementation reads the data from disk one chunk at a time. The operator is called to transform the chunks and/or compute a reduction object. The transformed chunks are then written back to disk and the computed reduction object is returned. The vector implementation applies the operator in the same manner regardless of the operator's implementation.

(ii) For a SPMD environment (Figure 4), the ANA runs in parallel in each process. Once the ANA in each process gives the operator object to the vector object, the vector implementation in each process applies the user-defined operator to only the local elements owned by the process. The intermediate reduction objects in each process are then globally reduced (i.e. using a single call to `MPI_Allreduce(...)`) and the final reduction object is returned. If the operator has a NULL reduction object, then no global reduction is performed and no communication is required.

(iii) In the client-server environment, the ANA runs only in the client process while the APP and LAL run in separate processes on the server. In this scenario, the operator object must be transported from the client to the server, where it is applied to the local vector data in each process. Then the reduction object is returned to the ANA on the client. The client-server configuration demonstrates the fundamental difference of this design from current approaches; the operator is taken to the data, the data is not moved to the operator. The application of an operator in a client-server configuration is involved and the reader is referred to [Bartlett 2001] for additional details.

There are several advantages to the RTOp approach. Specifically:

(1) LAL developers need only implement one operation — *`apply_op(...)`* — and not a large collection of primitive vector operations.
(2) ANA developers can implement *specialized* vector operations without needing any support from LAL maintainers. Note that *common* vector operators can be shared by the numerical community and need not be implemented from scratch by each set of developers of an ANA code.
(3) ANA developers can optimize time consuming vector operations on their own for the platforms they work with.
(4) Reduction/transformation operators are more efficient than using primitive operations and temporary vectors (see Section 6).
(5) ANA-appropriate vector interfaces that require built-in standard vector operations (i.e. axpy and norms) can use RTOp operators for the default implementations of these operations. In this way, some ANA developers may not ever need to work with RTOp operators directly in order to apply standard vector

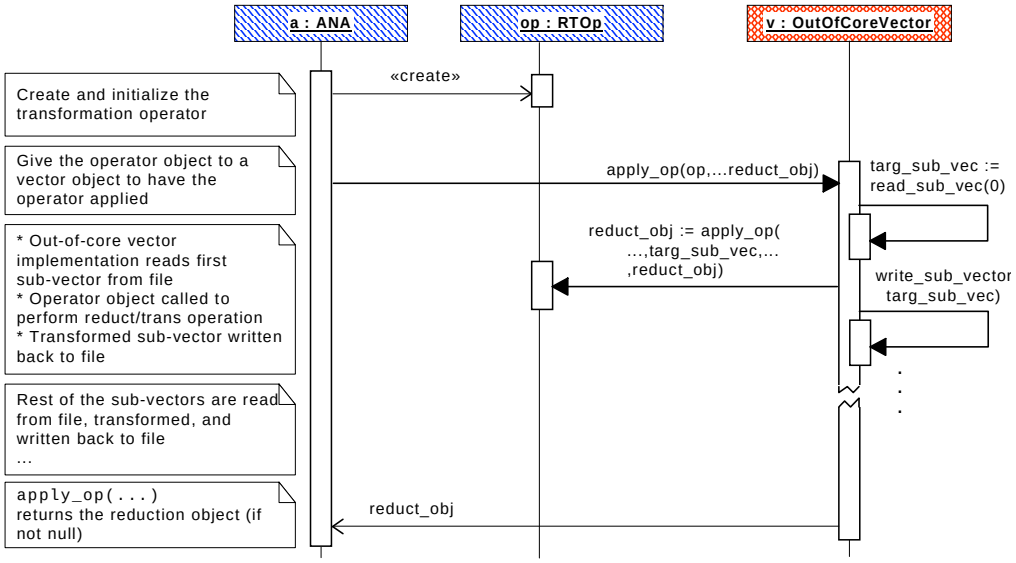

Fig. 3. UML interaction diagram : Applying a RTOp operator for an out-of-core vector

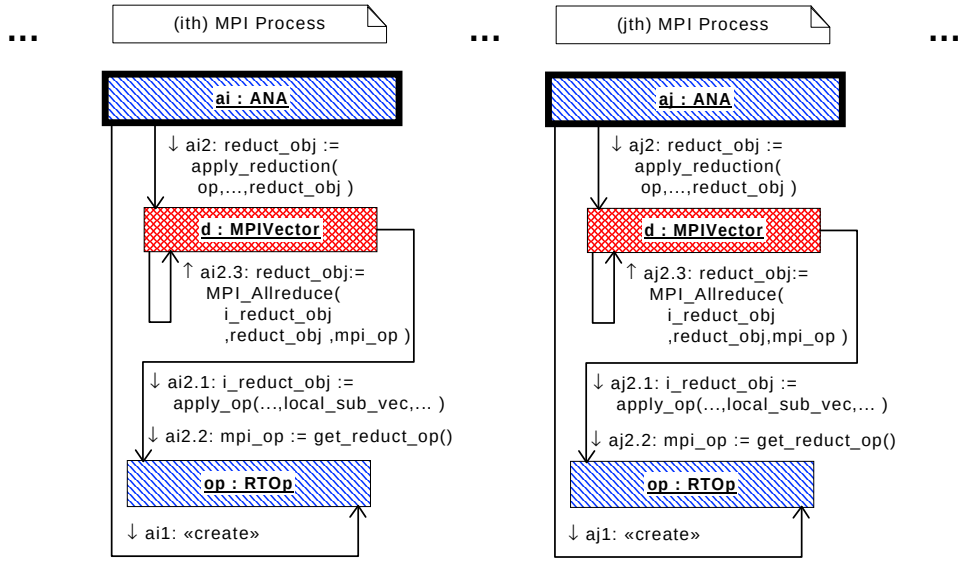

Fig. 4. UML collaboration diagram : Applying a RTOp operator for a distributed parallel vector

operations given a well written vector interface. In other words, the RTOp approach need not inconvenience ANA developers in any way.

5. AN IMPLEMENTATION OF POLYMORPHIC REDUCTION AND TRANSFORMATION OPERATOR OBJECTS IN C AND C++

The *RTOp* interface shown in Figure 2 can be implemented in a variety of different programming languages. Here we describe our initial implementatations C and C++. Because the design is based on object-oriented principles the obvious implementation language is C++. However, the use of C provides efficient support for the use of mixed languages and it makes the interface portable across many computer architectures. Also, because C is perhaps the most popular development language our approach may have a better chance of becoming a common specification. The basic reduction and transformation operator interfaces are fairly simple (i.e. single inheritance) and are therefore not difficult to implement in C. It must be emphasized that the only part of this implementation that must be adopted in order to realize the primary benefits of the new approach is contained in the single, relatively small, header file `RTOp.h`. The rest of the code at `WEBSITE` is strictly included for convenience and for demonstration purposes. The interfaces have been designed to be interoperable where operators implemented in C can be used through a C++ interface and visa-versa.

The most straightforward method to implement polymorphic objects and virtual functions in C is to reproduce the internal mechanics of a C++ compiler. To illustrate this technique, consider the C struct for reduction/transformation operators `RTOp_RTOp`:

```
struct RTOp_RTOp {
    void*              obj_data;
    RTOp_RTOp_vtbl_t*  vtbl;
};
```

In the above struct, `obj_data` is a `void` pointer to object instance specific data and `vtbl` is a pointer to a virtual function table. The address stored in the pointer `RTOp_RTOp::vtbl` uniquely determines the concrete type of the operator. The form of the data in `RTOp_RTOp::obj_data` must be compatible with the functions that are pointed to in the virtual function table `RTOp_RTOp::vtbl`. The struct for the virtual function table stores the pointers to the functions that are called at runtime, and creates the appropriate polymorphic behavior. For `RTOp_RTOp`, the virtual function table struct is defined as:

```
struct RTOp_RTOp_vtbl_t {
    const struct RTOp_obj_type_vtbl_t  *obj_data_vtbl;
    const struct RTOp_obj_type_vtbl_t  *reduct_vtbl;
    const char *op_name;
    int (*reduct_obj_reinit)( ... );
    int (*apply_op)( ..., RTOp_ReductTarget reduct_obj );
    int (*reduce_reduct_objs)( ... );
    int (*get_reduct_op)( ... );
};
```

In the above virtual function table, `apply_op` is a pointer to a function that will execute (10). Any reduction operation performed will be added to the `reduct_obj` argument. To simplify calling the `apply_op` function, a client can use the following function:

```
int RTOp_apply_op( const struct RTOp_RTOp* op
    ,const int num_vecs, const struct RTOp_SubVector sub_vecs[]
    ,const int num_targ_vecs, const struct RTOp_MutableSubVector targ_sub_vecs[]
    ,RTOp_ReductTarget reduct_obj )
{
    return op->vtbl->apply_op(op->obj_data,num_vecs,sub_vecs
        ,num_targ_vecs,targ_sub_vecs,reduct_obj);
}
```

A vector implementation invokes a vector operation on a set of sub-vectors `sub_vecs[]` and `targ_sub_vecs[]` given a reduction/transformation operator object `op` by calling:

```
RTOp_apply_op( op, num_vecs, sub_vecs, num_targ_vecs, targ_sub_vecs, reduct_obj );
```

In this way, the function `RTOp_apply_op(op,...)` acts polymorphically with respect to an operator object `op`.

The struct `RTOp_RTOp_vtbl_t` contains several other types of fields. The fields `obj_data_vtbl` and `reduct_vtbl` are actually pointers to two other virtual function tables of type `RTOp_obj_type_vtbl_t`. The purpose of these virtual function tables is to aggregate the methods needed to create, initialize, destroy, externalize and internalize the state of the operator and reduction objects. This design allows the same object structure to be used for both types of objects. The fields in the struct for this virtual function table are:

```
struct RTOp_obj_type_vtbl_t {
    int (*get_obj_type_num_entries)( ..., int* num_values, int* num_indexes
        ,int* num_chars );
    int (*obj_create)( ..., void** obj );
    int (*obj_reinit)( ..., void* obj );
    int (*obj_free)( ..., void** obj );
    int (*extract_state)( ..., void* obj, int num_values, RTOp_value_type value_data[]
        ,int num_indexes, RTOp_index_type index_data[], int num_chars
        ,RTOp_char_type char_data[] );
    int (*load_state)( ..., int num_values, const RTOp_value_type value_data[]
        ,int num_indexes, const RTOp_index_type index_data[], int num_chars
        ,const RTOp_char_type char_data[],void ** obj );
};
```

The ability to externalize and load an object's state as a set of arrays of simple data types (i.e. `value_data[]`, `index_data[]` and `char_data[]`) is essential for transporting and working with objects in a heterogeneous environment (i.e. client-server and heterogeneous MPI). The methods `extract_state(...)` and `load_state(...)` are esential for the use of MPI to perform global reductions efficiently

and to be able to transport RTOp operators over a network.

The remaining members in the struct `RTOp_RTOp_vtbl_t` are simple function pointers. The function pointed to by `reduce_reduct_objs` is used to perform (9). Finally, the function pointed to by `get_reduct_op` returns a pointer to another function that can be used by MPI as a user defined reduction operation with `MPI_Reduce(...)` and `MPI_Allreduce(...)`. This is the only place where `RTOp` specifically must adhere to the MPI standard. But this function could be used by any implementation to perform the needed intermediate reduction operations. When an operator does not return a reduction target object (i.e. performs a transformation only), some of the above function pointers can be `NULL` (see Appendix A).

As mentioned earlier, there is also a compatible C++ interface called `RTOpPack::RTOp`. There are several advantages of the C++ interface: operator subclass development is more straightforward, errors are handled with C++ exceptions (and not with tedious error codes), and memory management is easier due to C++ constructors and destructors. An excerpt from its specification is given below which shows similar syntax and functionality as the C specification:

```
namespace RTOpPack {
    class RTOp {
    public:
        ...
        virtual void apply_op( ... ,RTOp_ReductTarget reduct_obj ) const = 0;
        virtual void reduce_reduct_objs( ... ) const;
        virtual void get_reduct_op( ... ) const;
        ...
    };
}
```

A complete example program at `WEBSITE` includes an example vector interface, and different vector implementations (including a MPI implementation for SPMD programs). Also, several simple and unusual concrete reduction and transformation operator classes have already been written (in C) and tested that are available for general use which are described in the next section.

5.1 Examples of concrete `RTOp` operators

Source code (in C) for several simple and unusual concrete reduction and transformation operator classes is available for general use and to provide templates from which other operators can easily be built.

Table I shows some of the reduction/transformation operators that are already implemented. If a needed operator class uses an already implemented data type for its object instance data and reduction target object, then such an operator class is easy to implement. For example, consider the assignment-to-scalar transformation operator $z_i^0 \leftarrow \alpha, \text{for } i = 1...n$ in Table I. The header file and source files (stripped of comments) are simple enough and are given in Appendix A. This simple transformation operator class contains all the features needed for use in any computing environment. This subclass includes a constructor, destructor and other manipulation functions. The static implementation function `RTOp_TOp_assign_scalar_apply_op(...)` for this operator is trivial and only con-

RTOp operators	C Source files (*.c,*.h)
$z_i^0 \leftarrow \alpha$, for $i = 1...n$	RTOp_TOp_assign_scalar.*
$z_i^0 \leftarrow v_i^0$, for $i = 1...n$	RTOp_TOp_assign_vectors.*
$z_i^0 \leftarrow \alpha v_i^0 + z_i^0$, for $i = 1...n$	RTOp_TOp_axpy.*
$z_i^0 \leftarrow \text{random}(l, u)$, for $i = 1...n$	RTOp_TOp_random_vector.*
$z_i^0 \leftarrow v_i^0 / v_i^1$, for $i = 1...n$	RTOp_TOp_ele_wise_divide.*
$z_k^0 \leftarrow \alpha$	RTOp_TOp_set_ele.*
$z_i^0 \leftarrow -v_i^1 + \mu v_i^0 + \alpha v_i^0 v_i^1 v_i^2$, for $i = 1...n$	RTOp_TOp_mutiplier_step.*
$\alpha \leftarrow v_k^0$	RTOp_ROp_get_ele.*
$\alpha \leftarrow (v^0)^T v^1$	RTOp_ROp_dot_prod.*
$\gamma \leftarrow \max\{\alpha \mid v^0 + \alpha v^1 \geq \beta\}$	RTOp_ROp_max_step.*
$\alpha \leftarrow \sum_{i=1}^n v_i^0$	RTOp_ROp_sum.*
$\alpha \leftarrow \sum_{i=1}^n \left\{ log(v_i^0 - v_i^1) + log(v_i^2 - v_i^0) \right\}$	RTOp_ROp_log_bound_barrier.*

Table I. Selected reduction/transformation operator classes implemented in C

tains one significant executable statement (the for loop). This operator class uses a simple scalar object for its instance data (to hold α). This type is so common for both object instance data and reduction target objects that the virtual function table for it has been implemented in a separate source file so that it can be reused. The virtual function table RTOp_obj_value_vtbl is declared in the header file RTOp_obj_value_vtbl.h and its functions are defined in the source file RTOp_obj_value_vtbl.c. Since this operator class does not perform a reduction operation, it uses a predefined virtual function table RTOp_obj_null_vtbl which simply returns zero for the size of the object. With a NULL reduction target object, the last two function pointers in the struct RTOp_RTOp_vtbl_t are not needed and are simply made NULL.

Several virtual function tables for common data types have been implemented. However, the data types for some operators are unusual enough that these functions are implemented within the source file for the operator class. For example, see RTOp_ROp_get_sub_vector.c.

A considerable amount of boiler-plate code is required to create a new RTOp subclass. In order to ease the development process, a Perl script has been created that can be used to automatically create complete RTOp subclass implementations (in C) for many different types of specialized vector operations. Appendix B describes the mechanics of the perl script in more detail and shows the results of applying this script for two examples.

6. COMPUTATIONAL RESULTS

Conducting numerical experiments exclusively with low level linear algebra and some communication from reduction operations is somewhat predictable. However, we verify our design with two basic numerical experiments to validate an efficient implementation, and show better performance than any primitive stringing process.

In the first example, a test program based on a mock QMR algorithm is used to investigate the impact of using five primitive operations versus the all-at-once operator in (6). A total of 128 processors on CPlant [Riesen et al. 1999] was used in SPMD mode. The ratio of computation to communication was varied by manipu-

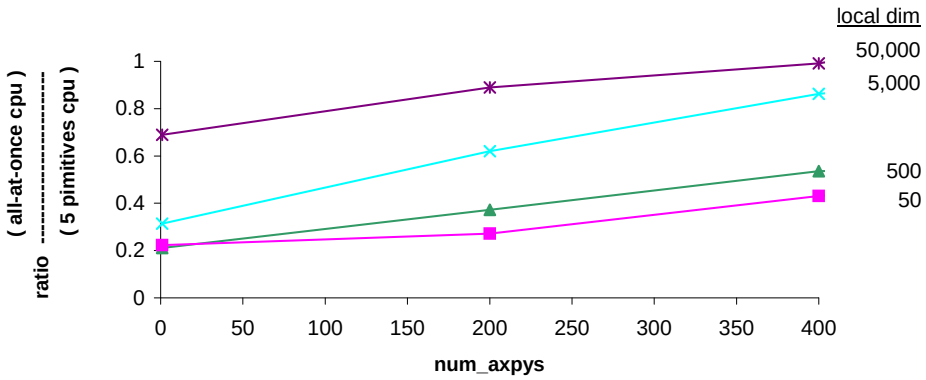

Fig. 5. Ratio of total process CPU times for using six primitive operations versus the all-at-once operator for the operatation in (6) (number of processes = 128, number elements per process = 50, 500, 5000, 50000 and `num_axpys = 1 ... 400`)

lating the number of local vector elements per process (`local_dim`) and the number of axpys per reduction (`num_axpys`). When (`local_dim`)(`num_axpys`) is sufficiently large, computation dominates and there is very little difference between the two implementations. However, when (`local_dim`)(`num_axpys`) is smaller, the all-at-once operator (with a single global reduction verses 5 global reductions) was noticeably more efficient. Figure 5 shows the ratio in runtimes for the two approaches. These results indicate that for nontrivial problem sizes the savings in runtime can be significant.

In the second example, the impact of multiple access of the same vector data and the creation of temporaries are investigated. The operation in (2) is used for the comparison for which the implementation using primitives is shown in (5). Figure 6 shows the ratio of CPU times for the all-at-once RTOp operator implementation versus separate primitive vector implementations. The C++ code is written using explicit loops and therefore removes any function call overhead that would otherwise have a dramatic impact for small vectors. There are two variants of the string of primitive vector operations implemented: one that uses preallocated temporary vectors (cached temporaries) and one that uses newly allocated temporary vectors for every evaluation (dynamic temporaries). Vector data is allocated using `std::valarray<double> v(n)` where `n` is the size of the vectors. Note that `std::valarray<>` is not supposed to call constructors on the vector data upon construction so in principle the construction of the vector object could be an $O(1)$ operation (this is not true for `std::vector<>`). This is strictly a serial program so there is no communication overhead to consider. The vector operators are performed several times in a loop and the ratios of runtimes are computed. The test program was compiled using GCC 3.1 under Redhat Linux 7.2 and run on a 1.7 GHz Pentium IV processor. As shown in Figure 6, even without the impact of multiple dynamic memory allocations, the implementation using the six primitive vector operations only achieved about 35% of the speed of the all-at-once operator.

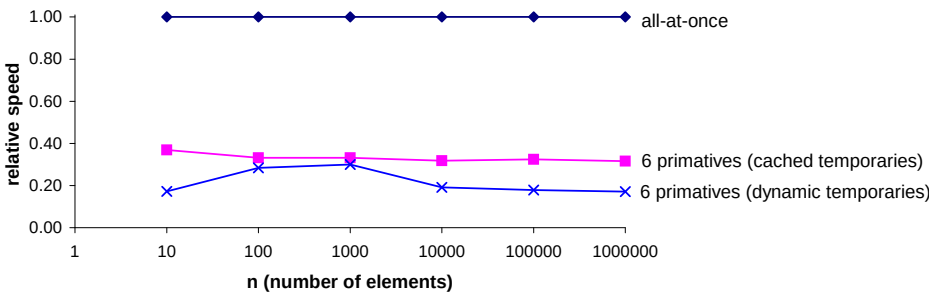

Fig. 6. Ratio of total process CPU times for using five primitive reductions versus the all-at-once operator for the operation in (2) and (5). The times for the primative operation approaches with cached temporary vectors and dynamically allocated temporary vectors are both given.

When naive dynamic allocations where used, the ratio of runtime dropped to about 20%. This example clearly shows the deterioration in runtime performance that the vector primitives approach can have over the all-at-once RTOp approach.

7. CONCLUSIONS AND FUTURE WORK

Growing complexities associated with computational environments, application domains, and data mapping place difficult demands on the development of numerical algorithms. The vector operator interface proposed here addresses these issues and allows the development of many types of complex abstract numerical algorithms that are highly flexible and reusable.

Advanced object-oriented design patterns were used to develop the RTOp interface and somewhat predictable numerical experiments demonstrate high efficiency in comparison to using a combination of primitives. Also, simple scalability tests confirm minimal serial overhead for large number of processors. Though the numerical efficiencies are noteworthy, development efficiencies and functionality provide the main advantages of this approach.

In summary, there are the five primary advantages to this approach:

(1) LAL developers need only implement one operation — `apply_op(...)` — and not a large collection of primitive vector operations.

(2) ANA developers can implement *specialized* vector operations without needing any support from LAL maintainers.

(3) ANA developers can optimize time consuming vector operations on their own for the platforms they work with.

(4) Reduction/transformation operators are more efficient than using primitive operations and temporary vectors (see Section 6).

(5) ANA-appropriate vector interfaces that require built-in standard vector operations (i.e. axpy and norms) can use RTOp operators for the default implementations of these operations.

A large set of vector operators are already available, but more significant is the flexibility to extend functionality. In addition, the extensions are independent of computer architecture and data mapping. By allowing the user to define reduction/transformation operators, all of the vector operations previously mentioned and many more can be implemented efficiently without requiring any temporary vectors.

The success of this approach relies on the adoption of a relatively small interface for reduction/transformation operators that is contained in `RTOp.h`. Future work will include the use of the RTOp approach in a client-server environment, which will specifically address development issues and bottlenecks related to distributed computing, heterogeneous networks and grid computing.

This design for vector reduction and transformation operators has been used to design very powerful ANA-LAL and ANA-APP (for nonlinear programming) interfaces in C++ called `AbstractLinAlgPack` and `NLPInterfacePack` respectively. These interfaces in turn have been used to upgrade a successive quadratic programming optimization package MOOCHO (a.k.a. rSQP++ [Bartlett 2001]) to allow fully transparent parallel linear algebra and arbitrary implementations of linear solvers (direct and iterative).

APPENDIX

A. IMPLEMENTATION OF "ASSIGNMENT TO SCALAR" `RTOP` TRANSFORMATION OPERATOR.

```
// ////////////////////////////////////////////////////
// RTOp_TOp_assign_scalar.h

#ifndef RTOP_TOP_ASSIGN_SCALAR_H
#define RTOP_TOP_ASSIGN_SCALAR_H

#include "RTOpPack/include/RTOp.h"

#ifdef __cplusplus
extern "C" {
#endif

extern const struct RTOp_RTOp_vtbl_t
    RTOp_TOp_assign_scalar_vtbl;
int RTOp_TOp_assign_scalar_construct( RTOp_value_type alpha,
    struct RTOp_RTOp* op );
int RTOp_TOp_assign_scalar_destroy( struct RTOp_RTOp* op );
int RTOp_TOp_assign_scalar_set_alpha( RTOp_value_type alpha,
    struct RTOp_RTOp* op );

#ifdef __cplusplus
}
#endif

#endif  // RTOP_TOP_ASSIGN_SCALAR_H

// ////////////////////////////////////////////////////
// RTOp_TOp_assign_scalar.c

#include "RTOpStdOpsLib/include/RTOp_TOp_assign_scalar.h"
#include "RTOpPack/include/RTOp_obj_value_vtbl.h"
#include "RTOpPack/include/RTOp_obj_null_vtbl.h"

static int RTOp_TOp_assign_scalar_apply_op(
    const struct RTOp_RTOp_vtbl_t* vtbl, const void* obj_data
```

```
    ,const int num_vecs, const struct RTOp_SubVector vecs[]
    ,const int num_targ_vecs
    ,const struct RTOp_MutableSubVector targ_vecs[]
    ,RTOp_ReductTarget targ_obj )
{
    RTOp_value_type   alpha = *((RTOp_value_type*)obj_data);
    RTOp_index_type   z_sub_dim = targ_vecs[0].sub_dim;
    RTOp_value_type   *z_val    = targ_vecs[0].values;
    ptrdiff_t         z_val_s   = targ_vecs[0].values_stride;
    RTOp_index_type   k;
    if( num_vecs != 0 || vecs != NULL )
        return RTOp_ERR_INVALID_NUM_VECS;
    if( num_targ_vecs != 1 || targ_vecs == NULL )
        return RTOp_ERR_INVALID_NUM_TARG_VECS;
    for( k = 0; k < z_sub_dim; ++k, z_val += z_val_s )
        *z_val = alpha;
    return 0;
}

const struct RTOp_RTOp_vtbl_t RTOp_TOp_assign_scalar_vtbl =
{
    &RTOp_obj_value_vtbl
    ,&RTOp_obj_null_vtbl
    ,"TOp_assign_scalar"
    ,NULL
    ,RTOp_TOp_assign_scalar_apply_op
    ,NULL
    ,NULL
};

int RTOp_TOp_assign_scalar_construct( RTOp_value_type alpha
    ,struct RTOp_RTOp* op )
{
    op->vtbl = &RTOp_TOp_assign_scalar_vtbl;
    op->vtbl->obj_data_vtbl->obj_create(NULL,NULL
        ,&op->obj_data);
    *((RTOp_value_type*)op->obj_data) = alpha;
    return 0;
}

int RTOp_TOp_assign_scalar_destroy( struct RTOp_RTOp* op )
{
    op->vtbl->obj_data_vtbl->obj_free(NULL,NULL,&op->obj_data);
    op->vtbl = NULL;
    return 0;
}

int RTOp_TOp_assign_scalar_set_alpha( RTOp_value_type alpha
    ,struct RTOp_RTOp* op )
{
    *((RTOp_value_type*)op->obj_data) = alpha;
    return 0; // success?
}
```

B. AUTOMATIC GENERATION OF RTOP SUBCLASSES IN C

To make the process of creating an RTOp C subclass easier and because there is boiler-plate code that is needed, a Perl script called `new_rtop.pl` has been implemented. This script automates most of the work required to create a new RTOp subclass. This script prompts the user for the answers to a set of questions about the operation being performed. In many cases, the output header and source files will be ready to compile and use. In other cases, the user will have to finish the implementation.

Many different types of specialized operators, including all of the example operators in (1)–(4), can be completely implemented with the script. Below, we show

the use of this script in generating the source code for C RTOp subclasses for the example specialized operators in (2) and (4).

B.1 Example transformation operator

The Perl script is first demonstrated on the transformation operator in (4). Note that all of the data required to perform the operation is contained in the four input vectors a, b, u and w. The only exception is the value of ∞ which may be platform dependent. Therefore, we will allow the ANA to define the value of ∞ as an operator object instance data member called `inf_val`. Before running the script, the vector arguments are ordered and mapped into the generic names $v^0 = a$, $v^1 = b$, $v^2 = u$, $v^3 = w$, $z^0 = d$ and then (4) is restated as:

$$z_i^0 \leftarrow \begin{cases} (v^l - v^2)_i^{1/2} & \text{if } v_i^3 < 0 \text{ and } v_i^1 < +\infty \\ 1 & \text{if } v_i^3 < 0 \text{ and } v_i^1 = +\infty \\ (v^2 - v^0)_i^{1/2} & \text{if } v_i^3 \geq 0 \text{ and } v_i^0 > -\infty \\ 1 & \text{if } v_i^3 \geq 0 \text{ and } v_i^0 = -\infty \end{cases} . \tag{11}$$

We will call this operator subclass `TOp_trice_diag_scal`. The interactive session with the script `new_rtop.pl` is shown below:

```
*******************************************
*** Create a new C RTOp operator subclass ***
*******************************************

1) What is the name of your operator subclass?
: TOp_trice_diag_scal

2) Is your operator coordinate invariant?
[y] or [n] : y

3) Give the number of nonmutable input vectors (vi, i=0...num_vecs-1)?
: 4

4) Give the number of mutable input/output vectors (zi, i=0...num_targ_vecs)?
: 1

5) Does your operator require extra data which is not in the input vectors?
[y] or [n] : y

Choose the structure of the data:
1: {index}
2: {value}
3: {value,index}
4: {value,value}
5: other
Choose 1-5? 2

Give name for {value} member?
: inf_val

6) Does your operator perform a reduction?
[y] or [n] : n
```

```
7.a) Does your element-wise operation(s) need temporary variables?
[y] or [n] : n

7.c) Give the C statement(s) for element-wise transformation operation?
   You can choose:
        Non-mutable operator data (don't change here)   : inf_val
        Non-mutable vector elements (don't change here) : v0, v1, v2, v3
        Mutable vector elements (must modify here)      : z0
? if(v3 < 0 && v1 < +inf_val )
?   z0 = sqrt(v1-v2);
? else if(v3 < 0 && v1 >= +inf_val )
?   z0 = 1;
? else if(v3 >= 0 && v0 > -inf_val )
?   z0 = sqrt(v2-v0);
? else if(v3 >= 0 && v0 <= -inf_val )
?   z0 = 1;
?

The implementation files RTOp_TOp_trice_diag_scal.h and
RTOp_TOp_trice_diag_scal.c should be complete!
```

After the script creates these files, they just need to be integrated into the build system (i.e. added to the makefile) and compiled. The only part of the implemented RTOp subclass that is more than just boiler-plate code is the loop that actually performs the element-wise transformation. Below is a snippet of code from the static function `RTOp_TOp_trice_diag_scal_apply_op(...)` for the loop that actually performs the user-defined element-wise transformation.

```
for( k = 0; k < sub_dim; ++k, v0_val += v0_val_s, v1_val += v1_val_s
    ,v2_val += v2_val_s, v3_val += v3_val_s, z0_val += z0_val_s )
{
    // Element-wise transformation
    if((*v3_val) < 0 && (*v1_val) < +(*inf_val))
        (*z0_val) = sqrt((*v1_val)-(*v2_val));
    else if((*v3_val) < 0 && (*v1_val) >= +(*inf_val))
        (*z0_val) = 1;
    else if((*v3_val) >= 0 && (*v0_val) > -(*inf_val))
        (*z0_val) = sqrt((*v2_val)-(*v0_val));
    else if((*v3_val) >= 0 && (*v0_val) <= -(*inf_val))
        (*z0_val) = 1;
}
```

The above code loops over a chunk of vector elements using BLAS-compatible strided iterators (of dimension `sub_dim` which are provided by the vector implementation) and performs the transformation operation. The generated source code can then be manually post-modified (and perhaps better optimized).

The developer of an ANA implemented in C++, for instance, can include the header file `RTOp_TOp_trice_diag_scal.h` and then a RTOp object for this transformation operator can be created, used and destroyed as:

```
#include "RTOp_TOp_trice_diag_scal.h"

...

void trice_diag_scale( const AbstractVector& a, const AbstractVector& b
    ,const AbstractVector& u, const AbstractVector& w, const AbstractVector& d )
{
```

```
    // Create and initialize the instance data for the operator
    const RTOp_value_type  inf_val = 1e+50;
    const RTOp_RTOp        trice_scal_op;
    RTOp_TOp_trice_diag_scal_construct(inf_val,&trice_scal_op);
    // Apply the operator to the existing vectors a, b, u, w and d
    const AbstractVector*  vecs[] = { &a, &b, &u, &w };
    AbstractVector*  targ_vecs[] = { &d };
    apply_op( trice_scal_op, 4, vecs, 1, targ_vecs, RTOp_REDUCT_OBJ_NULL );
    // Destroy the operator and clean up memory
    RTOp_TOp_trice_diag_scal_destroy(&trice_scal_op);
}
```

The above constructor and destructor are declared in the generated header file and are automatically implemented in the source file by the script. The above code snippet uses the C++ vector interface `AbstractVector` that is included in the example code. The `apply_op(...)` function simply calls the `apply_op(...)` method on the first `vecs[0]` object. Note that the order of the vector arguments `a`, `b`, `u` and `w` matches the order defined in (11). The ordering of the vector arguments must match and this order is determined by the developer that created the RTOp subclass.

B.2 Example reduction operator

The next example operator we consider is the reduction operation in (2). First we rewrite the operation in generic standard form as:

$$\alpha \leftarrow \{\max \alpha \ : \ v^0 + \alpha v^1 \geq \beta\}, \tag{12}$$

This reduction operation is more complex than the previous example transformation operation and requires a little more thought. If we can assume that $v_i^0 \geq \beta$, for $i = 1 \ldots n$ before going in, what the above reduction is really asking for is the minimum α_i where:

$$\alpha_i = \max((\beta - v_i^0)/v_i^1, 0).$$

The reduction operation (12) can then be reexpressed as:

$$\alpha \leftarrow \min\{\max((\beta - v_i^0)/v_i^1, 0), \text{ for } i = 1 \ldots n\}.$$

This reduction operation requires the scalar operator data β (`beta`) and produces the scalar reduction object α (`alpha`). To make this operator work correctly, we must initialize the reduction object `alpha` to a very large value. In this implementation we will assume that `1e+200` will be larger than any reasonable values of the reduction. In general, whenever using `min(...)` for the reduction of intermediate reduction objects, we generally want to initialize the reduction object to some large value before performing the first reduction.

We will call this operator `ROp_max_feas_step` and the following is the interactive session with the `new_rtop.pl` script used to create the implementation files.

```
*********************************************
*** Create a new C RTOp operator subclass ***
*********************************************

1) What is the name of your operator subclass?
: ROp_max_feas_step

2) Is your operator coordinate invariant?
[y] or [n] : y

3) Give the number of nonmutable input vectors (vi, i=0...num_vecs-1)?
: 2

4) Give the number of mutable input/output vectors (zi, i=0...num_targ_vecs)?
: 0

5) Does your operator require extra data which is not in the input vectors?
[y] or [n] : y

Choose the structure of the data:
1: {index}
2: {value}
3: {value,index}
4: {value,value}
5: other
Choose 1-5? 2

Give name for {value} member?
: beta

6) Does your operator perform a reduction?
[y] or [n] : y

Choose the structure of the data:
1: {index}
2: {value}
3: {value,index}
4: {value,value}
5: other
Choose 1-5? 2

Give name for {value} member?
: alpha

6.a) Does the reduction object require nonzero initialization?
[y] or [n] : y

6.b) Give the initial values for the reduction object data:
alpha ? 1e+200

6.c) Choose the reduction of intermediate reduction objects:
1: sum{value,value}
2: min{value,value}
3: max{value,value}
4: other
Choose 1-4? 2

7.a) Does your element-wise operation(s) need temporary variables?
```

```
[y] or [n] : n

7.b) Give the C statement(s) for element-wise reduction operation?
   You can choose:
        Non-mutable operator data (don't change here)   : beta
        Non-mutable vector elements (don't change here) : v0, v1
        Element-wise reduction data (must be set here)  : alpha_ith
? alpha_ith = ( beta - v0 ) / v1;
? alpha_ith = max( alpha_ith, 0.0 );
?

The implementation files RTOp_ROp_max_feas_step.h and
RTOp_ROp_max_feas_step.c should be complete!
```

The code snippet that loops through the elements and performs the reduction operation is contained in the generated static function `RTOp_ROp_max_feas_step_apply_op(...)` and is shown below.

```
for( k = 0; k < sub_dim; ++k, v0_val += v0_val_s, v1_val += v1_val_s )
{
    // Element-wise reduction
    alpha_ith = ( (*beta) - (*v0_val) ) / (*v1_val);
    alpha_ith = max( alpha_ith, 0.0 );
    // Reduction of intermediates
    (*alpha) = min( (*alpha), alpha_ith );
}
```

Since this is a reduction operator, the ANA code must create the reduction target object before it is passed into a vector object's `apply_op(...)` method. The follow code snippet shows how a ANA code might use this reduction operator and extract the value of the reduction.

```
#include "RTOp_ROp_max_feas_step.h"

...

RTOp_value_type max_feas_step( const AbstractVector& x, const AbstractVector& d
    ,const RTOp_value_type beta )
{
    // Create and initialize the instance data for the operator
    RTOp_RTOp        max_feas_step_op;
    RTOp_ROp_max_feas_step_construct(beta,&max_feas_step_op);
    // Create the reduction object
    RTOp_ReductTarget max_feas_step_reduct_obj;
    RTOp_reduct_obj_create(&max_feas_step_op,&max_feas_step_reduct_obj);
    // Apply the reduction operator to the existing vectors x and d
    const AbstractVector vecs[] { &x, &d };
    apply_op( max_feas_step_op, 2, vecs, 0, NULL, &max_feas_step_reduct_obj );
    // Extract the value from the reduction object
    RTOp_value_type  alpha = RTOp_ROp_max_feas_step_val(max_feas_step_reduct_obj);
    // Destroy the operator and clean up memory
    RTOp_ROp_max_feas_step_destroy(&max_feas_step_op);
    return alpha;
}
```

The above code snippet also uses the example C++ vector interface `AbstractVector` mentioned above.

That is really all there is to creating most new RTOp operators using the provided script. More details on the use of the script `new_rtop.pl` can be found in the help file `HowTo.CreateNewRTOpSubclass` at `WEBSITE`.

REFERENCES

Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenny, A., Ostrouchov, S., and Sorensen, D. 1995. *LAPACK User's Guide*. SIAM.

Balay, S., Gropp, W. D., McInnes, L. C., and Smith, B. F. PETSc, portable extensible toolkit for scientific computing (web site). http://www.mcs.anl.gov/petsc.

Bartlett, R. A. 2001. Object oriented approaches to large-scale nonlinear programming for process systems engineering. Ph.D. thesis, Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, PA.

Benson, S., McInnes, L. C., and Moré, J. TAO : Toolkit for advanced optimization (web page).

Blackford, L. S., Choi, J., Cleary, A., Azevedo, E. D., Demmel, J., Dhilon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walder, D., , and Whaley, R. 1997. *ScalLAPACK User's Guide*. SIAM, Philadelphia, PA.

Booch, G., Rumbaugh, J., and Jacobson, I. 1999. *The Unified Modeling Language User Guide*. Addison-Wesley.

Byrne, G. D. and Hindmarsh, A. C. 1999. PVODE, an ODE solver for parallel computers. *Int. J. High Perf. Comput. Applic 13*, 354–365.

Cai, X. 1999. Two object-oriented approaches to the parallelism of diffpack. `http://www.ifi.uio.no/~xingca/`.

Clay, R., Allan, B., Mish, L., and Williams, A. 1999. ISIS++ reference guide (iterative scalable implicit solver in c++) version 1.1. Tech. Rep. SAND99-8231, Sandia National Laboratories.

Clay, R. L., Mish, K. D., Otero, I. J., Taylor, L. M., and Williams, A. B. 1999. An annotated reference guide to the finite-element interface (FEI) specification : Version 1.0. Tech. Rep. SAND99-8229, Sandia National Laboratories.

Demmel, J. 1997. *Applied Numerical Linear Algebra*. SIAM.

Dennis, J. E., Heinkenschloss, M., and Vicente, L. N. 1998. Trust-region interior-point sqp algorithms for a class of nonlinear programming problems. *SIAM J. Control and Optimization 36,* 5, 1750–1794.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements fo Reusable Object-Oriented Software*. Addison-Wesley.

Gertz, M. and Wright, S. 2001. Object-oriented software for quadratic programming. `http://www.cs.wisc.edu/~swright/ooqp/`.

Gockenbach, M. and Symes, W. The Hilbert class library. http://www.trip.caam.rice.edu/txt/hcldoc/html/index.html.

Heinkenschloss, M. and Vicente, L. N. 1999. An interface between optimization and application for the numerical solution of optimal control problems. *ACM Transactions on Mathematical Software 25,* 2 (June), 157–190.

Heroux, M. A., Barth, T., Day, D., Hoekstra, R., Lehoucq, R., Long, K., Pawlowski, R., Tuminaro, R., and Williams, A. Trilinos : object-oriented, high-performance parallel solver libraries for the solution of large-scale complex multi-physics engineering and scientific applications.

J. J. Dongarra and J. Du Croz and S. Hammarling and R. J. Hanson. 1988. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft. 14*, 1–17.

Lumsdanie, A. and Siek, J. 1998a. ITL : the iterative template library. `http://www.osl.iu.edu/research/itl/`.

Lumsdanie, A. and Siek, J. 1998b. The matrix template library. `http://www.lsc.nd.edu/research/mtl/`.

Nocedal, J. and Wright, S. 1999. *Numerical Optimization*. Springer, New York.

Pozo, R. TNT: Template Numerical Toolkit. `http://math.nist.gov/tnt`.

Pozo, R. 1996. *LAPACK++ v 1.1: High Performance Linear Algebra User's Guide*. NIST.

Riesen, R., Brightwell, R., Fisk, L. A., Hudson, T., Otto, J., and Maccabe, A. B. 1999. Cplant. In *Proceedings of the Second Extreme Linux workshop*.

Sandia National Labs. 2001. ESI: Equation Solver Interface. `http://z.ca.sandia.gove/esi`.

Tuminaro, R., Heroux, M., Hutchinson, S., and Shadid, J. 1999. *Official Aztec User's Guide: Version 2.1*. Albuquerque, NM 87185.

Recieved: ???; revised: ???; accepted: ???